TYRON FRANCIS WWW.TFRANCIS.CO.UK

11 More complex devices

It all started in 1991. We spent the weekend in a house in the Canadian Rockies—Ian Witten, a computer science professor friend of mine joined us and all our hungry kids. The electricity had been switched off, and our first job when we arrived was to get power switched on—and to get the microwave oven working so we could cook supper. The microwave was a model aptly called the Genius, Panasonic model number NN-9807. Our task looked easy enough.

It took us about 45 minutes to get it switched on. Why did we find it difficult to switch a microwave oven on? What general lessons are there, and what can be done?

At first, I suspected that Ian and I might have been idiosyncratically bad at using microwave ovens; neither he nor I were expert microwave users. I spent the weekend trying to understand exactly how the Genius worked, and the next week I built a careful, accurate simulation of the Genius. I then did some experiments on other people—asking them to try to get my simulation of the microwave oven working, to try to do what Ian and I had failed to do.

The conclusion was that our difficulty was not unusual. The only advantage we had by being professors was that we didn't immediately blame ourselves for the problem, as it *couldn't* be our fault!

▷ The tendency of people to blame themselves for problems they experience with bad design is discussed in section 2.9 (p. 52), where we explore cognitive dissonance—a plausible explanation for the "it's *my* fault" syndrome.

Basically, the Genius would not start cooking. We presumed this was because it needed to know the time: it had some time buttons and was not displaying the time. When we pressed some of the minute and hour buttons, the Genius allowed us to get any number between 00:00 and 99:99 shown on its display. So we naturally thought that the clock accepted 24-hour times. Since it was late in the evening when we arrived at our friend's house, about 22:00 hours, we tried setting that time. The microwave seized up—you could say it really froze, a strange thing for a microwave oven to do.

We had to unplug to reset it and to carry on. Then we tried 22:05, then 22:15, ... and we tried later and later times as time went by, until finally we set the time to

1:00. (We had noticed that we were wasting our time telling the clock the correct time; we would pretend that it was 1:00, since the oven surely didn't really care.) The clock and the microwave then worked!

Having found one way of setting it, we soon realized that we had been misled by the clock. It was secretly a 12-hour clock, willing only to work when set to a time between 1:00 and 12:59, even though the display could be set to any number, including all the misleading 24-hour clock times we had been trying.

We then had a wager about what the user manual would say: would it say how to set the clock or wouldn't it? When we eventually found and read the manual, we agreed that we were both wrong: the manual did warn about the 12-hour clock problem but relegated the warning to a *footnote*!

We expected the clock to work one way but it didn't. Indeed, it gave us misleading clues that, as it could count so high, it obviously had to be a 24-hour clock. As long as we wrongly assumed that we knew how it worked, we would never be able to set the clock. Part of the problem was that we didn't think, "We're assuming how it works—let's check," because it seemed so obvious how it was working we did think we had assumptions to check.

There were at least two things wrong with the design. First, the clock should not have seized up when set to an "invalid" time (even one outside of 24 hours, like 66:66). Secondly, the user manual should have been a bit more helpful, if not more visible (perhaps there should be a label fixed on the side of the oven?).

▷ Anybody who writes a footnote in a user manual to give the user important information ought to tell the designer to fix the design—and make the footnote unnecessary! See section 3.3 (p. 65) where this idea is stated as a design principle.

Given that both of those design faults had been made, the user interface—the front panel of the microwave oven—should have been clearer. Or maybe the microwave should not need setting to any particular time before it works. You don't need to know that it is 10 o'clock in the evening to do 3 minutes of high-power cooking, so why does the microwave?

11.1 Starting to avoid design problems

When users don't understand how things work, they may be completely stuck, and *once* they get stuck—probably meaning they have a faulty idea in their minds about how the device works—there is little *they* can do to escape; it all comes down to the design, and how it handles error recovery. Or if the user has the *right* idea but the device just doesn't work like that, then they are stuck.

How could Panasonic, the microwave oven's maker, have avoided this design problem? Presumably the company tests designs on professional cooks and other potential users; whatever it did in the development process hadn't helped to fix the design problem. Indeed, several years later they were still selling microwave ovens with the same design: the manual of a later model just warned in much

bigger writing that is was a 12-hour clock. Clearly, even getting feedback from user problems isn't enough to help fix a bad design once it has gone into production.

Somehow designers need to evaluate their designs before they are committed to production. Unfortunately, it is rather too easy for designers to be biased when they try to anticipate how users will work with their designs. When designing the Genius digital clock, the designers probably assumed that everybody uses the 12-hour clock, and they probably tested the microwave on people who shared that assumption without thinking about it. It was an unspoken assumption. There would have been no design problem to solve because nobody would ever enter 10 p.m. as 22:00, and nobody in the design team or evaluation team would notice this as a potential flaw—until it was too late, that is, as the user manual's footnote makes clear. Noticing the design problem was came late in the process.

Some design errors cannot be found by using people, whether designers or test users. Yet you can be sure that there are users out there who will eventually stumble onto design problems. With the Genius my experiments suggested that about half the normal population of potential users would have had trouble in the afternoons; that's pretty bad, but not being able to use a microwave oven is hardly a disaster. With more safety-critical devices, say medical equipment designed for nurses to use, the chances of a user problem are lower (the nurses will be trained to use the devices), but the consequences of a design fault are much higher. Somehow we have to avoid preconceptions about use, and, in particular, preconceptions we don't even think about!

One approach to evaluating system designs is to carefully model users and to try to make realistic models of how they behave. Of course, this is very difficult. In the case in question how would we notice that we'd accidentally designed all the tasks to use 12-hour clocks? If, as designers, we are not aware that the 12-hour clock is a design issue, why should we build user models for testing purposes that (always or sometimes) use 24-hour clocks? The 12/24-hour question may seem pretty obvious in hindsight, but what of the design problems that we don't know about? What if we are designing a new product and *nobody* has any idea what the key issues are?

Although user models can be run relentlessly without a break and can therefore examine a large part of a user interface design, user models are fraught with difficulties. They may have systematic biasses, so, for example, parts of a design never get exercised. They may be incorrectly configured, so that timings and other measures are inaccurate and possibly misleading. In short, it seems easier to use real people directly to test a device, rather than to use them to build the models of what they would do. But real people on actual tests are they are expensive an slow, and it is very tedious to record what they do with a proposed design.

▷ See section 11.7 (p. 401) for ideas on spotting oversights in user testing.

Real people suffer from the same problems that user models do: we can't recruit the whole planet to do our system evaluation, so we inevitably miss out on some crucial behavior that somebody has. It would be easy to recruit ten people and for all of them to think the same way about clocks (particularly if they are our friends and relations, from the same office, or from the same country). If so, any study based on ten people would not be very insightful.

If we are designing our device specifically for young people, or the old, or the ill, then we should have proper concern for treating them ethically and legally—getting ten sick people to help us try out our design requires informed consent; getting children to help requires parental consent; and so on. Getting users who represent our target audience is tricky. If we are designing a device to give medical advice, and we are using people to test it, then there is a chance it will give misleading advice—perhaps because we haven't finished it, or because there is a fault in the user interface (that's what we expect—it's why we're doing the tests!) and the user simply gets the wrong advice. There are many complications to consider if we are serious about using people seriously for real device design!

It might seem counterintuitive, but a safer approach is to assume nothing about users and to get users who know nothing. Ignorant users might do absolutely anything, and if they behave randomly then they might assume that the clock was 24-hour; they might even assume it was 100-hour or 7-hour—and their problems would help designers discover new issues that nobody has yet thought of or had the patience to unearth. It could be useful to redesign to avoid the problems you discover with ignorant users. At least you should know the problems exist, so that palliatives can be worked out, say, by writing warnings in the user manual.

Once we see people making mistakes with 100-hour clocks, we can decide how to design for real humans. Maybe nobody would do this deliberately (maybe they would), but the design has to cope if anybody does it by mistake. We need to know the consequences of any user behavior, deliberate, ignorant, erroneous, or insightful.

The question is, where do we get such users to work with?

11.1.1 Using gnomes

There probably aren't any humans who can be relied on to behave truly randomly, so let's hire some gnomes. Gnomes are well known to be very ignorant and in particular to know nothing about microwave ovens, or anything else modern and interactive for that matter. They aren't worried about which buttons they press or whether pressing some buttons will break them (the ovens that is, not the gnomes). Apparently, our gnomes are so ignorant about microwaves that they might even try plugging and unplugging the microwave oven to see what happens—something Ian and I found out the hard way was a solution to getting it going after it had frozen up. (If we didn't let the gnomes do anything, then when the microwave freezes up, as it will, they would get *very* frustrated indeed, as gnomes do.)

For some applications, like security locks or things with safety latches, testing with gnomes would be a very good idea. If a gnome can break in or release the safety device, the security needs tightening.

A gnome is a metaphor for a user who embodies all possible wrong and right ways of using a system: gnomes can do anything. If we have a gnome to test designs, the designer cannot fall into the trap of assuming anything much and thereby being misled. This is a really important point. Other testing methods

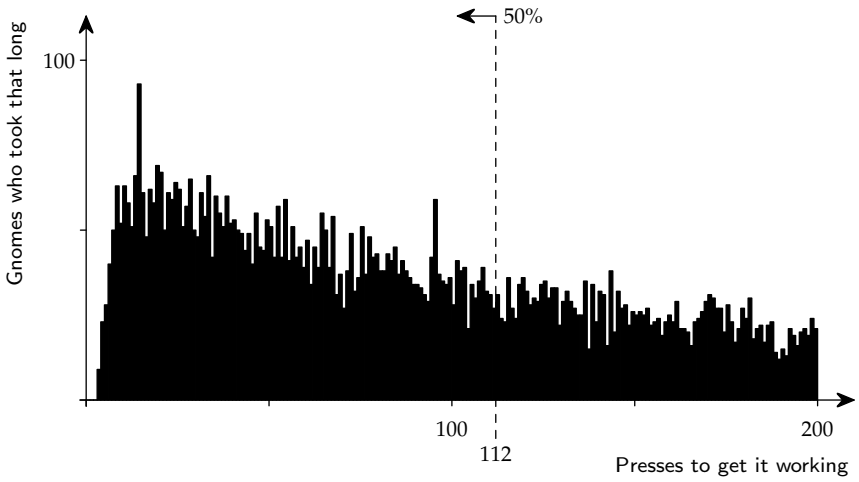

Figure 11.1: A bar chart showing how many gnomes managed to get the original Genius microwave oven to work in a given number of button presses. Some of the 10,000 gnomes used took more than 200 presses, and their data is not shown (which is why the 50% line seems too far to the right).

assume that the user (or perhaps a simulated user) knows what they are doing or that they make certain sorts of errors—in any case, generally that the user is doing some specific thing. A gnome approach makes no such assumptions; they model the user doing absolutely anything at all—and it is easy to have lots of gnomes working together or separately helping evaluate the device. In short, gnomes tell you, the designer, everything about a design, and in the end, when you add up all the figures, their performance gives a statistical overview of the design. Gnomes are also a lot faster than human users, so we get far more general results much sooner.

Let's sit a gnome down and get them to try to get the Genius to work. We'll count how many steps the gnome takes. Obviously sometimes gnomes will get lucky, and sometimes they will take ages. So we really need to hire lots of gnomes and average the results. I hired 10,000 gnomes and sat them down with a computer simulation of the Genius. They worked away, and I drew a bar chart of the results, shown in figure 11.1 (p. 371).

▷ Section 11.1.5 (p. 376) explains how to hire gnomes and how to put them to work.

Almost half the gnomes managed to get the microwave working in 100 button presses or less (in fact, the median is 112), some took over 200 presses to get it working, and one even took 1,560 presses! (Few humans would have the patience without doing more than unplugging it.) We are counting plugging the microwave in as one "press" because to get it to work after it has frozen up, it needs unplugging and plugging back in.

The gnomes seem to find this "simple" job really hard work. Yet if we asked the designer of the clock how many button presses it takes to set it, they might reply just *four* steps! That is, to get the clock to work after the oven's been plugged in, press [Clock] to enter the clock-setting mode, then press [1-hour], so it shows a valid time (namely, 01:00 o'clock), then press [Clock] again to start the clock running with that time. Easy—if you know how.

But my hired gnomes took 160 presses on average, a lot more than the designer's guess of 4. This huge discrepancy suggests that the design could be improved, or at least that the designer's optimism is unrealistic—or that we could train the gnomes better.

Let's look seriously at why the gnomes take so long compared to the designer's ideas. The Genius locks up when it is set to a time outside of the 1:00–12:59 window. There is absolutely no design reason in principle for a lockup. Removing the lockup (by redesigning the oven) dramatically helps the gnomes to be faster. Now, half the gnomes succeed in 77 or fewer presses, with an average time of 108 presses.

If we also changed the design so that impossible times, like 27:78, cannot be set, the gnomes get even faster, taking in average about 50 button presses to get the microwave going. Half of them will have got it working in 35 or fewer button presses—the tail of the graph. The number of gnomes who take more than 100 presses now has a factor of five fewer gnomes in it. That's a huge improvement. Figure 11.2 (p. 373) shows the bar chart based on the improved design.

Naturally we expect the unlucky gnomes to take longer on average than the designer's ideal or a typical human user, because, after all, gnomes don't know what they are doing. Nevertheless, our gnomes have helped us find a faster and easier-to-use design. The following table summarizes the results:

Design	*Average*	*Median*
Original Genius	161	112
Debugged not to freeze	108	77
Sensible design	49	35

So a little thought—motivated by random testing—lets us achieve a design that's on average about three to four times easier to use (at least for gnomes, if not for humans).

Now, when we humans use gadgets, much of the time we don't know how they work or what we are supposed to do; we're in much the same league of insight as the mindless gnomes—and a modified design, such as the one proposed here, that helps gnomes would also help us.

The improved design still supports all the original functionality (like cooking chickens), it just removes some design problems. Indeed, with human users, the faster design has the additional advantage of not allowing a user to display a misleading 24-hour time (like 22:02).

A random exploration of a design presupposes no specific knowledge of the user (or gnome). This has two advantages. First, a good designer ought to consider the possible wrong ways in which a design might be used. But there are infinitely many ways of being wrong, and a designer can only think of some of

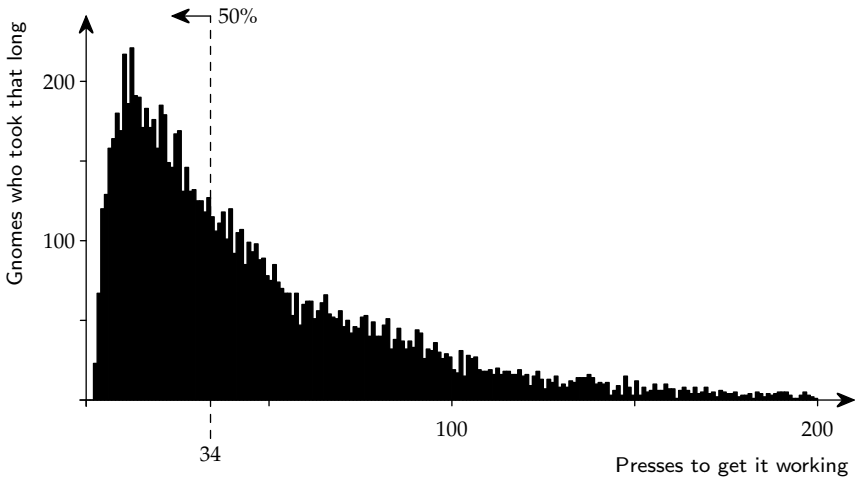

Figure 11.2: Bar chart showing how many gnomes managed to get the improved design of the Genius microwave oven to work in a given number of button presses. Compare with figure 11.1 (p. 371), which shows their performance on the original Genius. Now half the gnomes succeed in fewer than 34 presses, and only about 10% of the gnomes are taking more than 100 presses to succeed, whereas before half the gnomes took more that 112 presses. The improvement is clear from the way this graph tails off so quickly.

them. A random process, like our gnomes, however, embodies all possible wrong ways of using a system. Randomness is a remarkably effective way of testing out designs. After all, human users could only test according to their own few and fixed preconceptions. Moreover, if their preconceptions were the same as the designers, very little would be discovered about the design that the designer didn't already think they knew. Quite likely the original Genius design was made by a designer who didn't think in 24-hour times and they never thought to test for them.

So, although a gnomic "random user" is less efficient than a real human user, it cannot be tricked into guessing the designer's tacit assumptions. Gnomes are also a lot cheaper and faster than humans: being cheaper is good ecognomics, and you get them to work faster by using a metrognome (although I used a computer). This ease of testing with gnomes is their second advantage.

It is very interesting that a random gnome can set the microwave clock on average in 50 button presses, whereas Ian and I took far more. Our human intelligence was not helping us! We would have worked out what to do faster if we had simply tossed a coin to decide what to do next, because then we would have been working like gnomes.

11.1.2 Children as little gnomes

Certainly, being random is a better way of getting things to work than having the wrong ideas. This observation explains why children are so much better than adults at operating gadgets. Since they start off with no preconceptions, they press buttons pretty much at random. That approach gets results quickly, faster than a systematic approach that an adult would use.

So, we're not old and past it when we can't use gadgets. More likely, we know how they should work, but as they don't work that way, we get stuck. Children don't know how they work and they don't have preconceptions about "expensive" errors that scare us—we may be worried about breaking the expensive gadget or not using it "properly." But children, with no such worries, get along fine, and quite quickly understand examples of how the devices work. Once they have seen the devices work, not only does their confidence grow, but they can also learn from working knowledge. Put another way, using gnomes avoids all design assumptions. The gnomes are random button pressers. They do not work as we expect them to; they work (eventually) in all possible ways.

▷ Section 11.1.5 (p. 376) shows how gnomes can be programmed very easily.

The most likely reason why the Genius had its quirky design flaw was that its designers assumed that nobody would try a 24-hour time, and (if Panasonic did any user trials) the test users they picked also never thought to try a 24-hour time. The cultural assumptions caught them out, then. Since gnomes, like children, know nothing about what sort of clocks, 12 or 24 hours, we're supposed to use, they will check the design more thoroughly than any human process, whereas we know how devices should work, so we try to use them the right way. If a clock can count up to 99:99, it surely is a 24-hour clock. This is such a simple assumption, it will take you ages before you notice that you've assumed something plausible about the device that may not be true. The designers may never have thought that you'd think like this, and you haven't noticed you are thinking like it either.

▷ The problems, and benefits, of simple heuristics (like, if it looks like a 24-hour clock *it is* a 24-hour clock) is discussed in section 2.10 (p. 54).

Unfortunately, when our knowledge about the world is not reflected in the design of a device, we are worse off than not knowing anything. If we think we are right, but the device is designed otherwise, we will be stuck. One of our gnomes, or a child too young to know how the world should work, is prepared to experiment and play with "wild" ideas that we discounted as unrealistic and irrelevant.

▷ A general principle for a designer to follow to avoid this problem is to exploit affordance; see section 12.3 (p. 415). Also, design devices that do not have just one right way of use. Follow the principle of permissiveness, as raised in section 5.2 (p. 134). Section 11.4 (p. 387) discusses how permissiveness can make devices less susceptible to errors.

Children play to learn about the world. Adults stop playing because it is more efficient to use what knowledge they learnt through their early years of life spent playing. Adults also pay for the gadgets and they worry about breaking them. If

you are past playing with gadgets, then their odd design is frustrating rather than fun. The frustration itself makes it harder to enjoy using a device and persevering to find a solution! When you find yourself in this position, toss a coin or roll a dice—use some technique to make yourself approach the device like a gnome or a child. This will help you break out of whatever fixation you have become trapped by; with luck you'll be able to get on with your life quickly.

As always, good advice to a user can be rephrased as advice for a designer. Why not add a button to a device so that the device itself "rolls the dice" and presses a random button? Better still, why not bias the dice so that it only rolls to choose buttons the user hasn't tried pressing recently in the state? Then, if the user presses the PLAY! button (that is a less threatening button name than HELP), you could have a nice flashing display of all the currently untried options, and then the flashing slows down, cycling through the buttons, until—yes!—it's settled on suggesting Off! Well, maybe if the user switches the device off and on again, they will be more successful next time.

▷ Of course, we can do better than suggest the user presses a random button. See section 11.3.2 (p. 385).

Pressing a button at random, or to save the user the effort of having a button to make a random transition (so the user doesn't have to worry about working out a random action), can *help* the user, but sometimes a random action might be exactly what the user *wants* to do. They want to do something surprising, for fun, and what better than a random action? The Apple iPod Shuffle does just this: it can be set to play random tracks of music. Here, the "gnome" is a bit more intelligent than our gnomes—the iPod doesn't replay tracks it has chosen immediately. In fact, the idea of shuffle is that it first randomly shuffles the tracks, then plays from that list of tracks; otherwise, it would run the risk of repeating a lucky track too often or too soon. When it gets to the end of the list of chosen tracks, it reshuffles and starts again.

11.1.3 The bugs inside

One problem with the Genius is that the computer program inside it is childishly simple—perhaps another reason why children find it easy to use! More seriously, it has a bug.

There are four digits in the clock's time display, and simply, there are four buttons that adjust each digit. The 1-minute button increases the minutes; the 10-minute button increases the tens of minutes; the 1-hour button increases the hours digits, and so on. That's all the buttons do. The program is so trivial that 10-minute *always* increases tens of minutes, from 59 to 69 to 79. Neither 69 nor 79 minutes are valid times by anyone's clock. The programmer, perhaps pleased with the neat scheme of every button behaving exactly the same, forgot that the only button that should work so freely is the 1-minute. The 1-hour button should not change 2 to 3 if the 10-hour digit is already a 1, because we shouldn't be able to change the time from 12 o'clock to 13 o'clock because 12-hour clocks don't ever get to show 13. It is

absolutely trivial to get the programming right so that "times" out of the 1.00 to 12.59 window simply cannot be set. A modified design would then work like an analog wrist watch, where you simply can't set the time to something impossible.

11.1.4 Another microwave oven

To do all this we need a specification for the device, in this case another microwave oven. Once we have this, the definition of the device, its simulation, its usability analysis, and its user manuals in any language (and interactive help, if required), can all be worked on directly and efficiently in a suitable design framework.

If part of the design process suggests improvements, say, that the user manual has an obscurity, then it can be changed directly by modifying the specification; the new specification will update all other parts of the product, the analysis, the simulation, and so on. Importantly, the approach only has one definition of the device; thus changes—as occur during iterative design and product revision—immediately and automatically affect all parts of the development process: the analysis, the simulation, the help (and even the hardware).

Thus many components of a product are derived efficiently and automatically, almost at once. In normal design methods, there is a sequential (and costly) progression from specification, through fabrication, to manual writing, and finally usage. If any errors are detected, usually the whole process has to be started again, or when that is too expensive or would cause too much delay, one chooses to live with the errors—and maybe reprint the user manuals with warnings. In particular, only at later stages can usability problems be identified—but by then the product is already fabricated, and many of the usability insights would be very hard to take back to the specification, even it was still available.

To simulate the microwave oven, we can define it as a device in JavaScript using the framework from earlier chapters. We then add some gnomes.

▷ The original definition of the microwave oven we are using for this example appears in section 9.4.6 (p. 286). Using the same example show how a variety of design techniques handle the same device.

The microwave oven has a "touch pad" so we've changed the action words in the specification to describe the user's actions to be touching, rather than pressing. It would be a fun project to improve the English we will be generating with these boilerplate texts, but we won't go that way, except to point out that it isn't very hard to produce accurate and fairly clear if not perfect user manuals.

11.1.5 Gnome economics

Let's take, as an example, analyzing the user task of getting from power 1 state to power 2 state for this microwave oven.

The user may want to cook something but not at the current high-power setting. For the purposes of this chapter, analyzing just one task will be sufficient to show how it can be done. As before, rather than go to the trouble of getting a human user

(and wondering what sort of human, how familiar they are with microwaves, and so on), let's use a gnome again. Gnomes are cheap and they don't mind prodding things all day.

First we need a function that tells us which button the gnome should press. It looks worse than it is:*

```
function randomButton(d)
{ var r = Math.random();
  while( Math.floor(d.buttons.length*r) == d.buttons.length )
    r = Math.random();
  return Math.floor(d.buttons.length*r);
}
```

Generating random numbers is fraught with difficulties. It is *highly recommended* that you verify that your random number generator is working reliably, otherwise all your experiments will be suspect. Indeed, my first random number generator written in JavaScript failed some tests I tried on it; it is a worthwhile and often salutary experience to write program code to check your own program code—it helps find many errors.

A basic way to check that your random button pressing works properly is to do a few thousand trials and see how often each button would be pressed:

```
var testLimit = 100000;
var check = new Array(device.buttons.length);

for( var i = 0; i < check.length; i++ )
  check[i] = 0; // initialize array

for( var i = 0; i < testLimit; i++ )
  check[randomButton(device)]++; // count presses

for( var i = 0; i < check.length; i++ )
  document.write(check[i]+" "); // show results
```

The numbers should be approximately equal. Next you should do a χ^2 test on the results.

▷ The further reading for this chapter gives a good reference for random numbers and the relevant statistical tests.

We can use this `randomButton()` function immediately, simply to press buttons using the `press` function we defined for humans to use, as follows, where we try getting the gnome to press buttons ten times:

```
// test random pressing on the device; try 10 presses
for( var g = 0; g < 10; g++ )
{ press(randomButton(device));
}
```

* Numbers r returned from JavaScript's `Math.random()` are supposed to be uniformly distributed or, as most people say, "between 0 and 1," but more precisely $0 \leq r < 1$. Unfortunately some browsers occasionally return $r = 1$; in this case, the `while` loop repeats the call to get another random number.

Pressing buttons exactly like a human will mean that JavaScript will be updating the device display every time something happens—that's how `press` was defined in the framework; this is a waste of time (gnomes can't read), so we can speed up using gnomes by writing a "blind" press function:

```
function gnomePress(buttonNumber)
{ device.state = device.fsm[device.state][buttonNumber];
}

for( var g = 0; g < 10; g++ )
  gnomePress(randomButton(device));
```

For our next trick, it's useful to define a function to get state numbers from state names, to convert from names like power 1 to numbers like 4, which is the corresponding state number.

```
function toStateNumber(d, s)
{ for( var i = 0; i < d.stateNames.length; i++ )
    if( d.stateNames[i] == s )
          return i;
  alert("There is no state called "+s);
  return 0; // after error, don't return complete rubbish
}
```

Now we can try a "gnome trial" function, which takes the name of a state to start from and the name of a state to try to finish at:

```
function trial(d, start, finish)
{ d.state = toStateNumber(d, start);
  var f = toStateNumber(d, finish);
  var count = 0;
  while( d.state != f )
  { gnomePress(randomButton(d));
    count++;
  }
  document.write("Gnome takes "+count+" steps to get from "+
         start+" to "+finish+" states.");
}
```

▷ As noted in section 4.5 (p. 105), where we defined the function `plural`, the last function would generate better English if we wrote `... "Gnome takes "+plural(count, "step")+" to get from ...`

We start the gnome-simulating program in state power 1 and see how long it takes to get to power 2.

```
trial(device, "Power 1", "Power 2");
```

When I tried it, the gnome took 169 steps. Was this gnome lucky, or was the gnome a bad one who gave us an answer seemingly too hard to be reasonable? Or is the design bad? Until we do a lot more experiments, we can't tell whether we have learned more about the gnome or about the design.

One argument that the gnome was lucky is that we asked it to play with a device that happened to be strongly connected; if the device had not been strongly

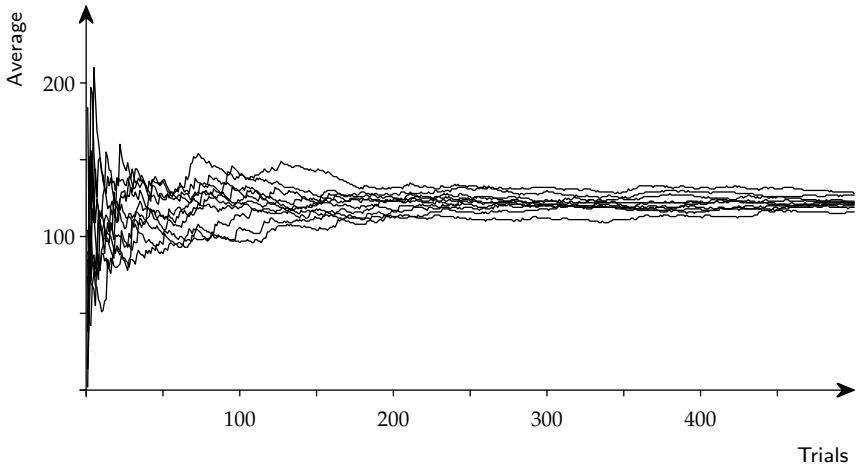

Figure 11.3: Ten gnomes testing a microwave oven to change its power settings. After time, the gnomes converge on taking about 120 button presses.

connected, the gnome would have run the risk of getting stuck somewhere (just as any human user would have risked getting stuck). The gnome wouldn't mind; it would just keep on pressing buttons—the program simulating the gnome would never give up, because the test `d.state != f` might always be true and it would go around the loop again and again! If the shortest path from start to finish states is ∞, no gnome will ever find a way of doing it; if the shortest path is finite but the device is not strongly connected, the gnome may or may not get stuck. If the device is strongly connected a gnome will eventually be able to get to the finish state (provided your random number generator is fair). Either you should check the device is strongly connected, or you should set a maximum limit on the number of times around the loop.

Whether the gnome was lucky or not, one useful thing we have learnt is that our design can cope with mad and relentless user testing for 169 presses. We should do some more testing, to check that the design is robust. (This sort of mindless but essential checking is something gnomes should be left to do, rather than using humans.)

We should try more gnomes and at least average the results to get better statistics. So we'll try some more serious experiments, hiring 10 gnomes at a time; we'll run 500 trials with each gnome and plot the results to see what we can learn: see figure 11.3 for the results. The graphs show that once we have 5,000 runs (ten gnomes each doing 500 trials), employing more won't add much more to what we already know. But for a more complex device than our microwave oven, this survey might be way too small. It's always worth drawing a graph to see how the numbers are working, and more informative (and a lot easier) than using statistics.

▷ In section 11.3 (p. 381) we work out what an infinite number of gnomes would average.

11.2 Drawing simple graphs in JavaScript

Figures 11.1 to 11.3 show graphs drawn from experiments with gnomes. Since graphs are a very good way of visualizing design parameters, I'll show you a very simple way of drawing graphs using JavaScript. If you want good graphics on web pages, server-side programming, if you want to use it, is more flexible; for instance, you can easily use PHP to generate graphics in any of a wide variety of formats. In general, drawing good-looking graphs is tricky unless you want to touch them up by hand, and you should use a standard graph-drawing package or general-purpose programs like Microsoft Excel or Mathematica, which can draw all sorts of fancy graphs with ease.

The simplest JavaScript trick for drawing a graph is to use HTML's `img` tag to draw a rectangle of the size you want. For example, `<img src=image height=42 width=73>` will draw the image with height 42 pixels and width 73 pixels. Just change the numbers to resize the rectangle.

Earlier figures in this chapter show how long it takes gnomes to get a device to work. For variety, we will draw a graph of the shortest path costs: if the device is doing something, how long (how many steps) does it take to make it do something else? An easy-to-use device will have lots of low costs; a hard-to-use device will have lots of high costs. Some devices might have lots of low costs, and one strange high cost, which perhaps deserves your closer inspection, unless you expected it. Overall, the shape of the graph of costs—it's general shape—is what's interesting.

```
function barChart(d)
{ var apsp = shortestPaths(d);
  var n = apsp.length; // number of states
  var chart = new Array(n);
  for( var i = 0; i <= n; i++ )
    chart[i] = 0; // initialize the bar chart to zeros
  for( var i = 0; i < n; i++ )
    for( var j = 0; j < n; j++ )
      if( apsp[i][j] <= n )
        chart[apsp[i][j]]++; // count lengths
  for( var i = 0; i < chart.length; i++ )
    document.write("<img src=blob.gif width=5 height="+(1+chart[i])+">");
}
```

In the JavaScript above, the variable name `apsp` just means "all pairs shortest paths" and is the matrix of shortest path lengths between any pair of states: so `apsp[i][j]` is the least number of steps a user could take getting from state `i` to state `j`.

▷ The code for `shortestPaths` was given in section 9.6 (p. 297).

To draw a graph of these numbers, create an array `chart` and initialize it to zero. Then the work is done in the two nested `for`-loops that examine every element of `apsp` and count how often each value occurs. Since `apsp[i][j]` can be infinity in devices that aren't strongly connected, count entries $\leq$ `n`—unless it's ∞, *no* path can be longer than the number of states.

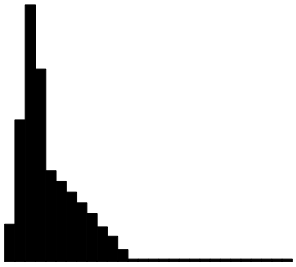

Figure 11.4: A simple bar chart showing the distribution of path lengths for the JVC HR-D540EK PVR. With more programming, you can get axes and other useful information included—our simple JavaScript hasn't shown that the horizontal axis is the cost of getting from one state to another (a number, for this device, ranging from 0 to 11) or that the vertical axis is how often any particular cost appears. For this device, the most common path length (the mode) is 2, occurring 192 times. Without the axes, it isn't obvious that the leftmost bar shows how often a zero path length occurs: for any device it's going to be the number of states, since you can stay in any state by doing nothing. Other uses for path data are shown in box 9.2 (p. 310).

Having collected the information, a simple loop draws the graph using the image-scaling trick:

```
for( var i = 0; i < chart.length; i++ )
  document.write("<img src=blob.gif width=5 height="+(1+chart[i])+">");
```

The `1+` in the height expression ensures we get at least a 1 pixel-high line for the graph axis—a 0 pixel-height line would not show anything. Figure 11.4 (p. 381) shows the sort of result you might get.

11.3 Markov models

Gnomes are a fun and convenient way to test how well product designs work, and the statistics we need to understand what they do is easy—we can get away with drawing graphs, taking averages, and so on.

A more sophisticated approach is to use Markov models, which are a mathematical way of exploring what gnomes would do. Markov models allow us to work out, in effect, what an infinite number of gnomes would do. We can use Markov models to get more reliable results, and they are *considerably* faster than running an infinite number of tests on gnomes! Using our techniques, we'll also soon show ways to halve the difficulty of using a microwave oven, even if we are as ignorant as gnomes are about microwaves.

Andrei Markov did a lot of work on random processes, and Markov's techniques lend themselves to mathematical treatment. The details are quite tricky, but that doesn't matter, because we can ask someone else to do the math (and then

Box 11.1 Matrices and Markov models The stochastic matrix of section 11.3 (p. 381) is just the same as the cost matrix of section 9.6 (p. 297), except that the costs (0, 1 or ∞) have been replaced with probabilities. Instead of it being the case that a button press takes one step to get to another state in the finite state machine, the stochastic matrix expresses the fact that there is a probability the user will press the button.

Since each row represents all the states a user can end up in, whatever they do, each row adds to 1.

Suppose the stochastic matrix is called S and **v** is a vector representing the current state the device is in; it's called the state vector. Then $\mathbf{v}S$ will represent the state the device is in after the user has done one thing. More precisely, $\mathbf{v}S$ will give a distribution; it will show the probability that the device is in each state after the user's action. The beauty of this is that we can see further ahead easily: $\mathbf{v}S^n$ tells us the state distribution after exactly n button presses.

With not much more effort, from **v** we can work out the expected number of states the user will have visited after n presses; this is how we worked out the cost-of-knowledge graph in section 5.3 (p. 144), graphed in figure 5.11 (p. 144).

Standard things to do with Markov models are to find out the expected number of actions to get to a state, which in user interface terms represents how hard doing something is on average, or to find out the long-term behavior of a device.

Usually the transition matrices are considered for the *whole* device, but we can also take the matrices for each button considered as its own finite state machine. This generates the button matrices. If we have buttons [On] and [Off], we can easily find the button matrices On and Off that represent the transitions that these buttons can achieve.

If the device is in state **v**, then **v**On is the state it is in after [On] is pressed. Matrices easily allow us to work out theories of the device behavior; in this example (even with such little information) it is likely that On times the matrix Off is equal to the matrix Off—in words, pressing [Off] after pressing [On] has the same effect as pressing [Off] directly. The matrix equation says that this is true in every state. Since matrices are very efficient for calculating this is a good way of discovering many deep properties of a user interface.

write a program for us); we can use the results. Fortunately, if you have Markov modeling added to your design framework, you won't need to understand technical details.

To do a Markov analysis, first we must convert the device's definition into a so-called stochastic matrix. Here it is displayed in traditional mathematical notation:

$$\begin{pmatrix} 3/5 & 1/5 & 1/5 & 0 & 0 & 0 \\ 2/5 & 2/5 & 1/5 & 0 & 0 & 0 \\ 2/5 & 1/5 & 0 & 1/5 & 1/5 & 0 \\ 2/5 & 1/5 & 1/5 & 0 & 0 & 1/5 \\ 2/5 & 1/5 & 0 & 1/5 & 1/5 & 0 \\ 2/5 & 1/5 & 1/5 & 0 & 0 & 1/5 \end{pmatrix}$$

▷ This is exactly the same sort of matrix as we encountered in section 9.6 (p. 297), except that now the entries are probabilities rather than button costs.

▷ Box 11.1, "Matrices and Markov models" (p. 382) explains more about how the matrices work.

Each row gives the probability that the user—or gnome!—will change the state of the microwave oven; thus, if the device is in state 1, the gnome will change it to state 2 with probability $^1/_5$ (i.e., first row, second column). There are five buttons for the gnome to choose from, and with probability $^1/_5$ it chooses the button that changes the state to 2. Sometimes there are two buttons that change the current state to the same state, hence the $^2/_5$ probabilities. For now, the assumption is that each button on the device is pressed with equal probability (there are five buttons, so all the probabilities are so-many fifths); the user interface simulation can give empirically-based probabilities, which we will use later. This probability matrix can be fed into our analysis.

The math on this matrix gives a result of 120. This is what thousands of gnomes averaged at, but it was faster to work out using a Markov model (even though we needed to be good at math; for many people it's easier to program the gnomes than to work out the Markov models).

It is interesting to choose the probabilities differently. Rather than assuming that each action or button press is equally likely, as above, we can work out the best way of achieving the task and set *those* button presses to have probability 1. Now our gnome is behaving like one that knows what it's doing.

If we rerun a Markov analysis on this matrix of "perfect knowledge," we should get back the shortest ways of doing anything. Indeed, the answer here is 2. Which emphasizes just how bad 120 is.

Evidently, the more knowledge about the device design, the easier a device is to use. Difficulty of use can be plotted against knowledge in a graph that combines the complete randomness of a gnome with increasing amounts of design knowledge. The graph should obviously speed up between 120 (the result of ignorance) to 2 (gained with the aid of perfect design knowledge). Indeed, this is what we see in figure 11.5 (p. 384).

11.3.1 Alternative designs

A gnome presses buttons at random and takes 120 button presses, but a knowledgeable designer who knows what to do would take only 2 button presses to do the same thing. Gnomes takes a long time because they know nothing. The designer is efficient because they have privileged design knowledge. Surely there must be a way of telling gnomes *something* so that they can do a bit better than pure chance?

Of course, if we work out what hints to give gnomes, and they work well enough, then we can give the same hints to human users.

One reason that the gnomes are so bad is that they even try doing things that don't work at all. They might try pressing a button that leaves the microwave oven in the same state. Now, whatever they are trying to do, doing nothing cannot help. So, let's run the experiments again but this time getting the gnomes to only press buttons that are guaranteed to do something.

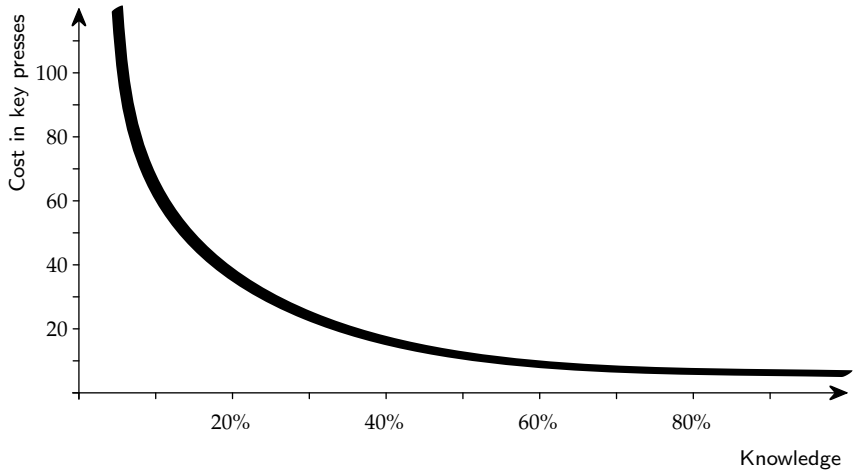

Figure 11.5: The more you know, the easier it gets. When you know 100%, you are as good as an expert, here taking only 2 presses to change power settings on the microwave oven. If you know nothing, you may take a very long time, but on average you will do it in 120 presses.

In human terms, we could imagine that we redesign the microwave oven so that its buttons light up when they are active, or equivalently that they are dark when they are not going to change the current state. In a sense, this makes the device a polite device: it tells you when it can do things for you. A polite device tells the user that a button will do something or that a button won't do something *before* the user wastes time finding out.

▷ The opposite of politeness, rudeness, is discussed in section 6.3.4 (p. 185).

We could either write a slightly more sophisticated gnome-simulating program (press buttons at random, but out of the ones known to change the current state) or run a Markov model on a revised matrix of probabilities. When we do, the answer for the power-changing task drops from 120 to 71.

We can modify the original `trial` code to work out the times the gnomes take with the original design and with a modified design where they avoid pressing buttons they know will do nothing.

```
function trial2(d, start, finish)
{ d.state = toStateNumber(d, start);
  var f = toStateNumber(d, finish);
  var count = 0, newcount = 0;
  while( d.state != f )
  { var oldstate = d.state;
    gnomePress(randomButton(device));
    count++;
    if( d.state != oldstate ) newcount++;
  }
```

```
  document.write("Original device: gnome takes "+count
         +" steps to get from "+start+" to "+finish+" states.");
  document.write("Modified device: gnome takes "+newcount
         +" steps to get from "+start+" to "+finish+" states.");
}
```

Actually, the code here has the gnomes pressing buttons whether they make any difference or not—but the variable `newcount` only counts the presses that do change state. Surprisingly, this simple change to the gnomes' behavior helps them achieve *any* task; in particular, it almost makes the task we were considering almost twice as easy. It's worth doing.

The general design rule is that a device should provide the user with feedback about the future, not just its present state. Here, if the device knows a button is not going to do anything, it says so!

In fact, exactly the same idea is used in graphical user interfaces on desktop computers: you will find menu-items dimmed out when they aren't going to work. Now we know why.

11.3.2 Even cleverer buttons

▷ Section 11.1.2 (p. 374) suggests playful buttons, and we've just suggested smart buttons that light up if they might be worth pressing (in the last section). Section 9.6.4 (p. 304) suggests lighting buttons to speed up the user according to the Hick-Hyman law.

There are very good reasons why both of these design ideas can help users. Can we do any better? The two small innovations we have suggested are based on knowing what the user wants to do:

- The user is stuck. Show them something random; it may help.
- The user wants to do something. Show them which buttons do something.

▷ If we *really* know what the user wants to do, then the device may as well do it. This idea leads on to invisible devices and incidental interaction; see chapter 12, "Grand design."

So far in our simulations, the gnomes have been pressing buttons equally often on average. But if one button stands out from the others, we'd want our gnomes to prefer pressing it, and it should be pressed more often—that is what human users would tend to do. We might get a few real people to use a prototype of a device, to get real button press probabilities, and we could then be precise about the probabilities to use—generally, without doing experiments with users, we won't know what anyone is trying to do with any reliability or accuracy.

If we assume we have already got the probabilities for each button in an array `device.probpress`, it's easy to write a revised random button function:

```
function distributedRandomButton(d)
{ do
  { var r = Math.random(), cp = 0;
    for( var b = 0; b < d.buttons.length; b++ )
    { cp = cp+d.probpress[b];
      if( cp > r ) return b;
    }
  } while( b >= b.buttons.length );
}
```

As before, this code loops if the random number generator incorrectly returns exactly 1—it goes around and tries another number. The code can be made faster by pre-computing an array of cumulative probabilities rather than repeatedly calculating `cp` each time the function is called. Of course, the code won't work correctly unless the button probabilities `probpress` add to 1.0, as probabilities should. More details of algorithms with random numbers can be found in Knuth's *Art of Computer Programming*—see this chapter's further reading, p. 403.

▷ Section 11.3.1 (p. 383) explores concrete ideas for improving button pressing probabilities. Making some things *too* easy may not be a good idea—see section 5.4 (p. 145).

A user is looking at a device with fifteen buttons that could achieve something. Either the user knows what to do, or, in the most general terms, this user will have to *search* for the right result.

Computer scientists know a lot about how to make searching more efficient. Obvious suggestions are to put the possible target states of the search into a sequential list, to put them into an alphabetically sorted list, or to use more interesting techniques, such as binary search, hashing, trie search, or tree search.

▷ The impact of alternative kinds of search techniques (for a fax) are explored in section 5.5.5 (p. 153).

We can only usefully put states into order if we know what the user is searching for. If it's a command name, then putting the commands into order will help. But in general, we really have no idea what the user is searching for—and the user may not know until encountering it. If they don't know what the command is called, then putting them into alphabetical order in itself doesn't help much—it's no better than being in a linear list in *any* order.

Even if we have no order (such as the alphabet) that can be used, we can still help the user search systematically. There are two important and very general approaches to searching:

Breadth-first search Try to keep as close as possible to the starting state, and then move out to new states, increasing the distance from the start as slowly as possible. This approach is good if we have no idea where the goal is.

Depth-first search Go as far as possible in any direction before trying alternatives. This approach is good if we think we are going in roughly the right direction.

Of course, in both breadth- and depth-first search, we don't repeatedly search places we've already searched.

To help a user do *any* search, we need to know a few things: where they started from (for instance, off or standby), what have already been visited, and whether they want depth or breadth-first search. Typically a user won't know what the technical terms breadth-first and depth-first mean, so we could have buttons like "help me broaden my search" and "help me search here."

When the user presses one of these buttons, the device searches for an unvisited state that is either close to standby or close to the current state but farther than it from standby.

▷ Close states are readily found using the shortest paths algorithm from section 9.6 (p. 297), perhaps using more appropriate costs than button presses, as suggested in section 9.6.4 (p. 304).

Since there are lots of possible states to try next, the device will score them as it searches and then present the user with the button press that scores highest. I've tried these techniques on web sites, and they are very successful in helping people find what they wanted—even though the web site has no idea what they are looking for! But users don't go round in circles any more, and they don't miss nearby pages they might otherwise have ignored.

To see whether these ideas would improve your device, tweak the approach to suit the sorts of things users do with your device and experiment to see how well it works—then do some more tweaking to the scoring system to get more useful results.

11.4 Human errors

The previous sections suggested how we can get design insights by comparing random behavior to "perfect" behavior. We imagined that a designer knows the best way to use a device, and we drew a graph of cost against how good the user (or gnome) was compared to optimal use. We showed that we could "guide" random behavior by having more interesting buttons and feedback to the user of what does what.

Gnomes are not people, and we are designing for people, not gnomes. How do people differ. More precisely, what do we know about people that we can apply to design?

We know that people tend to make certain sorts of systematic error. We could translate the way users work into Markov models, because we know that if a user has done x, they may now be more likely to do y. Markov models can handle this, but it gets very difficult to provide all the data they require. If, on the other hand, we stick to simple, first order models, we only need to know—or guess—the probabilities of pressing buttons regardless of whether the user has done x or y. This is much easier, and the results, while perhaps less accurate, are at least uniformly inaccurate, whereas an incomplete model might be more accurate *sometimes*, but you may not know when.

That is, if you train your sophisticated model to reflect real user behavior, unfortunately you run into exactly the same problems that all user testing has: it takes a very long time to get sufficient, and sufficiently broad, data to be useful. Of course, you can quickly get an impression, but almost by their very nature, errors are surprising—they are unlikely—so you need even more data to model errors well.

Rather than trying to model anything a user might do, it's more productive to model specific things based on how we know people behave.

- ▷ Task/action mappings, discussed in section 9.6.5 (p. 311), help us explore how a user does a task successfully.

One way to look at user errors is to consider their effect on degrading from "perfect" performance. We can easily find a shortest path (using task/action mappings) and treat this as the model of error-free behavior. Then what might a user do?

- They may make slips and randomly fall off the path anywhere. We have already modeled this sort of error very well.
- They may do things in the wrong order.
- They may miss steps on the path but carry on doing the right things (though probably now on the wrong path).
- They may forget some initial steps, missing out the "preparation," but then follow the rest of the path correctly.
- They may not get to the end of the path, stopping short of performing all the necessary actions. That is, they behave as if they have finished, when there is yet more to do. These are completion errors.
- They may follow a correct path for a different task. These are transfer errors. The user has transferred the wrong actions (which were right in a different context) to the intended task. The different task may or may not be one from the same device—it might be a transfer error like trying to use a Nokia device like a Samsung.
- If two paths start the same way (or, put formally, if they have a common prefix) then the user may continue down the preferred path—often, the more frequently used one—even if this is not the path they intended to go down to start with. More generally, if multiple tasks share parts of their paths (they have common subsequences), it's possible that users will start off doing what they intended, including the shared part of the path, but then follow the wrong branch after the shared part. This is a capture error. The more familiar, more frequently performed actions have "captured" the user.
- Users may not stop when they have otherwise completed the task. This is an overrun error.

Box 11.2 Ways to avoid human error Humans make errors for all sorts of reasons, and psychologists distinguish different sorts of error: slips, lapses, mistakes, and variations thereof. The suggestions for further reading give pointers to more details, but what we want to do is *avoid* errors, however they are classified, and then—given that we can't always avoid errors, recover from them or reduce their consequences when they do occur.

Don Norman suggests six important strategies for reducing error. Exactly how these principles are applied depends on the context; the important point is to think through the options carefully.

- Make information visible, so the user can *see* what to do next. We've talked a lot about using indicators as one way to do this; see section 11.3.2 (p. 385) for example.
- Simplify the design to reduce reliance on the user's memory.
- Use affordances (see section 12.3, p. 415)—use "natural" mappings, simple relationships between controls and effects.
- Use "forcing functions" to guide users. Constraints make it hard to do the wrong things; forcing functions make it impossible. For example, the button that fires the rocket has a cover; the user is forced to lift the cover before launching the rocket.
- Assume that errors *will* occur and design accordingly. Provide techniques for error recovery.
- Finally, standardize actions, layouts, displays, symbols, and so on; make systems consistent.

▷ See the boxes 6.1, "Syringe pumps" (p. 168) and 11.3, "Generating manuals automatically" (p. 397) for example errors. Errors are also discussed in section 11.4 (p. 387), and box 5.7, "The principle of least effort" (p. 147).

▷ We considered certain classes of overrun errors, namely accidentally repeating the last step, and how to assess the design's impact on them in section 10.4 (p. 339).

Armed with these ideas, we want to reduce the *consequences* of errors for users, and we want to increase the *robustness* of the devices we design—we want to "design errors out." However, given that errors will always occur sooner or later, we also want to reduce the *probability* that an error is made in the first place, and we want to reduce its *persistence* once it's made.

Somewhat out of our hands is training users properly and giving them full experience of the device. The most important thing is for a device to be simple, straightforward, even elegant in its design, that is, easily learned; after that, a device should provide good feedback to users, so that they can monitor what it is doing and check whether this matches their expectations.

The list of errors above strongly suggests where to emphasize feedback: for example, after a common subsequence of actions, a user will have to make a choice, whether to continue what they intended or continue an erroneous task they have just been captured into. The branch of paths after a common subsequence of actions is easy enough to identify automatically; an example (potential) design error that should be flagged is that indicators do not change at the branching state.

- ▷ Sometimes feedback to the user is ambiguous: users won't know whether they have done anything, let alone the right thing! See section 10.3.2 (p. 334). Sections 11.5 (p. 392) and 11.6 (p. 396) provide interaction programming techniques to provide accurate training material, so users know how to interpret feedback. Section 10.7.5 (p. 358) showed how to ensure that the user gets wide enough experience of a system in their training.

Error consequences can be improved by adding undo, for instance. Robustness can be improved by introducing more permissiveness into the design. We can also improve things by making the state of the device clearer—then users may notice problems sooner, before they are beyond undo or other recovery. Making the state clearer will help the user do their work with fewer errors.

Being more specific about reducing consequences requires us to know more about the costs of errors for users. For example, not switching your television on—you've sat down, but it's still off—is not as costly an issue as leaving your money behind in a cash dispenser (sometimes called an automatic teller machine, or ATM), and that, in turn, is not as costly an error as leaving your cash card behind (which might allow a thief to remove any amount of cash from your account).

That brief analysis suggests that given a choice, the more costly errors for the user should be arranged, by redesigning the device, to come earlier in the sequence. Indeed, this is what we find on cash machines: if they are well designed, users have to remove their cash card before taking their money. With this design, users are very unlikely to leave their card behind—but they may still leave their money behind.

This redesign works in harmony with a quite different explanation of completion errors. A user went to the cash machine with the task "get cash" uppermost in mind. With the better design, the user can only complete the intentional task *after* having already removed the card—the device has forced the user to do something that otherwise might be overlooked if cash was dispensed first.

We can invent ways of measuring robustness. One simple measure is the number of ways a user can achieve a task. If there is one way to do a task, then any error makes the task fail; the more ways, the less likely any error (of any of the sorts listed above or any other sort of error) will make the user fail.

We can measure this simple robustness by counting the different number of paths through the device to achieve the same objectives. For the sake of argument, we consider all tasks possible on a device to be equivalent to considering all pairs of states: every task then has a start state and an end state. We count how many and how long paths are between all pairs of states and take averages to get a measure of robustness for the "average task." (Counting paths can be done using a depth-first search.)

For our microwave oven, we get the graph shown in figure 11.6 (p. 391). The figure shows quite clearly that for this device any task ending in a power setting—that is, actually cooking—is a lot less robust (in the sense we are measuring robustness!) than either quick defrosting or using the clock. Of course, we would get more profound results if we weighted averages according to our knowledge of the device. For example, since the user cannot leave a microwave oven cook-

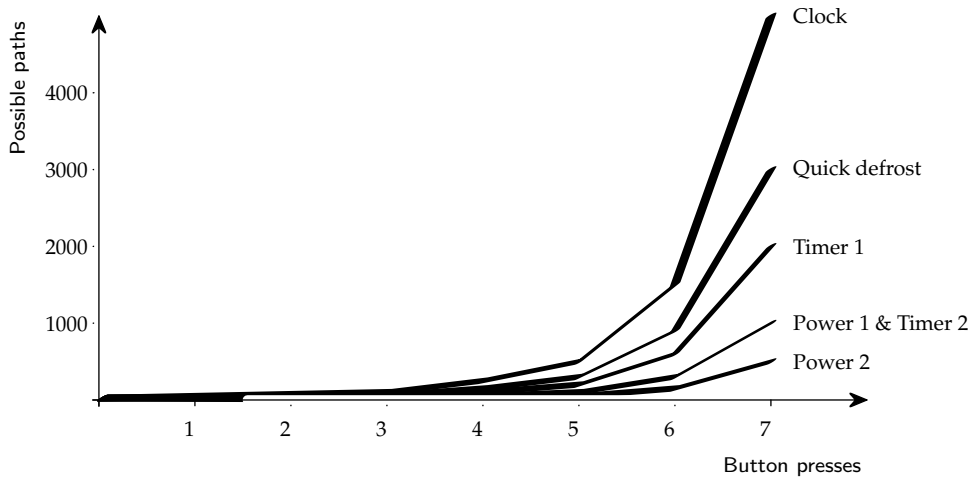

Figure 11.6: One visualization of robustness; here, measured as the number of possible ways of reaching a given state, plotted against the number of button presses needed to reach it. The numbers are averaged over all possible starting states. The larger the number, the more permissive or robust the target state.

ing forever, most tasks the user will do will not start from any cooking states. For simplicity, we assumed all starting states were equally likely—intriguingly, the graph for this device looks the same and doesn't change the ordering of the results (though with different numbers) when we redo the analysis assuming never starting in a power state.

Robustness against error is *not* the same as safety. Cars have numerous safety features, including the following: you cannot change from park to drive without pressing the brake pedal; you cannot remove the key unless you are in park; in some manual cars, you cannot start the engine unless the clutch is depressed—to avoid risking starting the car when it is in gear, and, incidentally, to make it easier to start (the starter motor won't be turning over the gear box's cold oil).

A car is not robust in the sense that there are lots of ways for the users to achieve their goals; the point is to ensure that users can only achieve goals safely. Some possible states and transitions are designed out; the user simply cannot do them. A "robust" car might allow you to start the engine when the car is in drive without your foot on the brake pedal, but a safe car would prohibit this error. Some errors you want to forgive the user, some (like starting a car when it is in gear) you want to block, before something worse happens—though occasionally it is good practice to start in gear, for instance on steep hills. Designers have to make tricky tradeoffs.

▷ A measure of robustness is the size of strongly connected components; we used the farmer's problem as a motivating example to redesign devices in section 10.2.1 (p. 329).

Figure 11.7: User error is defined by design error—illustrated here by the JVC UX-NB7DAB CD/radio. Evidently, the manufacturers are aware of the design defect as the CD tray door has a warning label "*CAUTION DO NOT TRY TO FORCEFULLY OPEN AND CLOSE THE DOOR BY HAND TO PROTECT FROM MALFUNCTION*." The label has been made removable because it isn't visually aesthetic. However, if the label was removed the design would not become aesthetic anyway: it would then have a *hidden* interaction defect and be worse. The photograph shows what happens if you drop a CD when attempting to change one: the door may close with the CD trapped, covering the very button that is supposed to be used to open and close the door. Now you have to fiddle with the trapped CD—or force the door!

Finally, people can also make sophisticated errors involving reasoning. They may have, in some sense, quite the wrong idea of the tasks they are trying to achieve. In controlling a complex system like a nuclear power plant, indicators tell the user all sorts of things about the overall situation. The user then has to reason about what to do to avoid a catastrophe. Many things can go wrong—not least being that the power station itself overwhelms clear thinking with far too many alarms.

11.5 Automatic, correct help

There is a legal requirement that descriptions of products should correspond with the products themselves: under the UK Sale of Goods Act 1979 (as amended by the Sale and Supply of Goods Act 1994 and the Sale of Goods (Amendment) Act 1994) products should be "fit for purpose" and should correspond with the description of them. Thus it is the (UK) law that user manuals are correct—or, if we take a weaker view, that the manufacturer at least knows what the correct description is, so that some appropriate description, but in any case truthful, can be written for the user. Other countries will have similar laws.

Although our device definition is very basic, it can be used to generate useful help for the user or for technical authors (technical authors can at least start from an accurate draft of the manual).

We now define a function `help` that explains the shortest path (the least number of button presses) to get from any state to any state. The definitions given below can be adapted straightforwardly to provide clearer help if "buttons" aren't actually pressed (maybe, for example, they are knobs that have to be twisted).

The device might have an interactive feature, so pressing a button gives help—perhaps showing it in a display panel. If so, it might be defined partly as follows, making use of the current state: here is a small part of the microwave oven's manual:

To get from the device Power 1 to Power 2:
Press [Time]
Press [Power]

▷ Program code to generate manual entries like this is based on finding shortest paths, discussed in section 9.6 (p. 297).

We'll need this fact later. Note that the best way of getting from Power 1 to Power 2 takes two button presses, as we realized in the previous section.

Ideally one would write more sophisticated manual-generating programs to generate better natural language. In particular, straightforward parametrization of the program would allow equivalent manuals to be generated in any appropriate language.

If we developed a typographical style for user manuals, then all devices processed in the framework would be able to use that style. Also, one could generate interactive HTML manuals for the web, and then the user could also follow hypertext links to learn the exact workings of the device.

We can print an entire manual just by finding the best way to get from each state to every other state. It's still a bit long and boring, but it starts off like this:

To get from the device Clock to Quick defrost:
Press [Quick defrost]
To get from the device Clock to Timer 1:
Press [Time]
To get from the device Clock to Timer 2:
Press [Time] Press [Time]
To get from the device Clock to Power 1:
Press [Time] Press [Power]
To get from the device Clock to Power 2:
Press [Time] Press [Time] Press [Power]

This doesn't provide a particularly easy or enjoyable read, but it is a complete and correct manual that a technical author could start to work from. However, it is well worth generating, even if nobody reads it. It is quite possible that some things are not possible for a user to do, so trying to generate a manual that explains

how to do everything at least checks that the user can do anything they want to. We should certainly write a program to generate this manual and run it on any proposed device; if the program "gets stuck" then the device has problems.

We might prefer to typeset the user manual in a different format. Here is an extract of one generated for the JVC PVR that starred in the last chapter:

> ...
> If you are playing a tape, but have paused it, you can:
> Press [Play] to play a tape
> Press [Operate] to off, with tape in
> Press [Forward] to fast forward
> Press [Rewind] to rewind a tape
> Press [Stop/Eject] to on, with tape in
> If you are playing a tape fast forward, you can:
> Press [Play] to play a tape
> Press [Operate] to off, with tape in
> Press [Pause] to pause playing a tape
> Press [Stop/Eject] to on, with tape in
> If you are playing a tape fast backward, you can:
> Press [Play] to play a tape
> Press [Operate] to off, with tape in
> Press [Pause] to pause playing a tape
> Press [Stop/Eject] to on, with tape in
> ...

This form of manual is very easy to make interactive. If we recoded it in HTML, it could look like this:

```
...
<a name="pause">If you are playing a tape, but have paused it, you
can:</a>
<blockquote>
  Press <a href="#play">[Play]</a> to play a tape<br>
  Press <a href="#offtapein">[Operate]</a> to off, with tape in<br>
  ...
</blockquote>
...
```

A user can press buttons in the manual itself—it's just a web page, viewed in a browser—and it would work just like the device it described. You can get nice-looking buttons relatively easily by using cascading style sheets. Better still, you could use images for the buttons.

For many devices, whether a PVR or microwave oven, a user's tasks won't be to get from a known state to another state, but simply to get to the desired state, regardless of the initial state. We can generate a manual for this sort of use.

To represent a device in an unknown state, we represent its possible states as a set, and we define a function to find out what set of states the device will be in after a given sequence of button presses; it involves some fun programming. A

breadth-first search can then be used to look for unique states. Then, by defining some routines to explain things in (for instance) English, we can print out the sequences of button presses to get to each state. We now have the user manual that tells a user how to do anything regardless of what the device is doing to start with. Notice how short it is. Perhaps because of its brevity, we can get some interesting design insights straight from it.

> Whatever the device is doing, you can always get it to
> Clock by pressing [Clock].
> Quick defrost by pressing [Quick defrost].
> Timer 1 by pressing [Clock], then [Time].
> Timer 2 by pressing [Clock], [Time], then [Time].
> Power 1 by pressing [Clock], [Time], then [Power].
> Power 2 by pressing [Clock], [Time], [Time], then [Power].

This time I had some fun making the English a bit nicer. It says, "press a, b, *then* c"; the data for this microwave oven never needed the fancy "press *twice*" feature. But making a manual look nice helps—and is more fun to program.

Looking at these instructions, it looks like the clock button ought to have been called [Reset]. If so, note that you can still get to the quick defrost state by pressing it (that is pressing the original button [Clock]) first, then the [Quick defrost] button.

We might think that if such a manual is "good," what would a device look like that this manual was the *complete* explanation for? To find out, all we need to do is change the English-printing routine to one that goes back to the device specification and sees which parts of it are used and which are not. Here's the result, summarized as a table:

States \ —*Buttons*—	[Clock]	[Quick defrost]	[Time]	[Clear]	[Power]
Clock	✓	✓	✓		
Quick defrost	✓	✓			
Timer 1	✓	✓	✓		✓
Timer 2	✓	✓			✓
Power 1	✓	✓			
Power 2	✓	✓			

▷ Compare this table with the state transition table used for the same device in section 9.2.2 (p. 277).

We could draw this table in many different formats, which might be a good idea to try for much bigger systems—when size and readability become more serious issues.

Look at the ✓ entries in this table: these are the only parts of the microwave oven's state machine specification that the user manual required. The [Clear] button doesn't seem to be helping much—there are no ticks in its column! Our generating a manual and then automatically going back to the specification has exposed potentially bad design elements. If this sort of manual is a good idea, then the [Clear] button as presently defined is a design feature that needs better justification.

Here is an extract from a manual generated for a more complex system, the JVC PVR from earlier chapters:

> In the following states (play a tape fast forward; pause playing a tape; play a tape fast backward) you can press [Play] to play a tape.
>
> If you are playing a tape, but have paused it, additionally you may:
> —Press [Forward] to fast forward.
> —Press [Rewind] to rewind a tape.
>
> In the following states (play a tape fast forward; play a tape fast backward) you can press [Pause] to pause playing a tape.
> —If you are playing a tape fast forward, you cannot do anything else.
> —If you are playing a tape fast backward, you cannot do anything else.

This chunk of the manual would be "inside" a section explaining how the buttons [Operate] (the badly named switch on/switch off button) and [Stop/Eject] work, since they can be used at any time.

Many other sorts of manuals can be generated too, and by creating them systematically with programs we can guarantee their correctness. We can also use the technique of going back from a good manual to reappraise the specification. After all, if we have a good user manual, then the bits in the specification that aren't apparently needed are immediately suspicious features.

11.6 Tools to support user manuals

We can use automatically generated sketch manuals to help technical authors to write better and more accurate user manuals. We would write a program like the one we used above to generate accurate but perhaps not best English into a database. Imagine that the technical author's tool would look a bit like a multiwindow word processor: one window would show the generated manual—correct but maybe not good English—and another window the current bit of the manual that the technical author is writing. The technical author would rephrase the English in their window, and perhaps group sections of the automatic manual together if they could think of a more concise way of describing them. This manual writing tool would have a button that simply prints all the nice text the technical author has written. And to be really useful the following features would be needed:

- Warn if there are any sections of the automatically generated manual that the technical author has missed using.
- If the device design is changed, reuse the tool to find automatically all of the manual sections that may need changing. The technical author might decide against changes, but they would certainly like to be shown all sections that possibly need revising. So each section the technical author writes needs a checkbox to indicate they've been "checked out" as OK when the specification changes.

Box 11.3 Generating manuals automatically Good typographical layout of manuals is essential. Here is an example of how a badly set manual contributed to death: in June 1988 a crowded 18:47 train into the Paris railway station Gare de Lyon crashed with 56 dead and many more injured . The accident inquiry noted that many factors led to the brake failure: an air brake was faulty, the train had already had an emergency stop in another station, and a radio alarm was faulty. But the inquiry noted that the maintenance manuals were particularly complex and they highlighted a misleading error in their layout.

The relevant part of the French manual starts off, "Plusieurs véhicules sont bloqués, le mécanicien: s'assure que ce blocage n'est pas la conséquence de la fermeture d'un robinet d'arrêt de la conduite générale situé avant la partie du train bloquée . . . " Jacques André has translated and given the structure of the manual (see further reading, p. 403, for the reference):

```
The driver checks:
1st CASE: x x x
          x x x
2nd CASE: y y y
          y y y
          In both cases the driver should
          z z z
Then, the driver restarts the train.
```

The situation in the Gare de Lyon accident was covered by Case 1; unfortunately the driver failed to notice the "in both cases" instructions (which in the original French were not as obvious as here), as they were indented and looked like part of Case 2 procedure. The driver omitted to do *z z z*, and the train crashed. The line beginning "In both cases" should not have been indented; indeed, given its importance, the first line might have been better rephrased along the lines of, "*After* checking both cases (below) the driver should *z z z*."

Had the manual been generated automatically, if anybody had thought of a better layout to clarify any part of the manual (and it is a big manual, so there are lots of cases where improvements might have been noticed), the improved layout scheme would have been used automatically everywhere. One idea to improve layout or structure, or whatever, then automatically becomes applied *everywhere*, consistently throughout the manual, at no further cost to the designers or technical authors.

In other words, automating a manual greatly improves its reliability, because mistakes (here, layout mistakes) can be detected and eliminated efficiently and systematically.

11.6.1 Minimal manuals

All devices have user manuals. Sometimes the manuals are printed paper booklets, sometimes they are just a few words on the back of the packaging, and sometimes they are on a CD or available on some web site. Sometimes, indeed, explicit user manuals don't exist because of oversight, costs, or because the designers thought the design was "obvious." But in all cases, user manuals exist in principle, whether or not the manufacturers have gone to the trouble of providing them.

If a user manual is complete and honest, then it stands to reason that the longer it is. the more the user has to learn before understanding the device. Conversely,

the shorter a user manual, while remaining faithful to the design, the simpler the system must be to understand, if not use.

- We can easily measure the length of basic manuals that have been generated automatically. Indeed, you don't actually need to generate the manual to know how long it is; it's easy to write programs to estimate the length. The estimate may be awry for all sorts of reasons, but as the device design is changed, changes in the manual length can be very informative for the designer. A new feature that doubles the length of the basic user manual should be immediately suspect (unless it is *very* useful)!

- Experiments show that when user manuals are rewritten to be shorter, they are more effective, even if the device design does not change. One of the few outcomes of direct commercial value from interaction research is the minimalist technique of developing manuals: it is cheap and easy to do, and it effectively helps users perform better.

- Typically, modified designs that result in shorter manuals will be easier to use. Why not, then, set up an automatic process to generate good-enough user manuals (if we are only interested in their length, they don't need to have a lot of detail), then modify designs so that the automatically generated manuals get shorter?

There are different sorts of manual. Do not confuse manuals and technical documents written to help designers or to help the design process with manuals written to help the user. What helps a user decide to buy one product or another, say, by looking up functionality they want, is very different from what helps a user do what they want to do with the product.

▷ Section 8.8 (p. 259) discusses spanning trees and the algorithms for finding them as a way of generating manuals.

Users typically want to focus on their actual tasks and activities. The approach for designing manuals for this purpose is called minimalist instruction.

Instead of telling a user how to use all the functions of product, tell them how to do useful tasks—and identify what these tasks or activities are. Get rid of clutter about what the device is for, assume that users already have domain knowledge—knowledge about what they got the device to do in the first place.

When users do use a device, they make mistakes. Therefore an important part of a minimal manual (*especially* a minimal manual, which by its nature doesn't cover everything) is how it supports the user successfully recognizing and recovering from errors.

Rarely do users read manuals from start to finish. Manuals should therefore be designed so that users can jump around to what they want to do, from moment to moment. In fact, manuals should be hypertext, or web-based documents.

11.6.2 Validating scenarios, manuals, and legal documents

Any scenario (a story about a device and its use) or any user manual, even a draft, runs the risk of becoming obsolete or having errors introduced because the designer has modified the device design—perhaps "just a little improvement" and they didn't bother to tell the author, or perhaps they made 37 improvements and bug fixes, and the author only managed to sort out 35 of them before the designer did some more tweaking. In any case, the technical author needs all the help they can get!

We needn't confine the technical author to a secondary role in the design process. The author might have some good ideas about the design they'd like the designer to adhere to. The author might be writing sales material, offering a device with certain nice features. The author might be writing a legal document, claiming that the device does certain things in certain well-defined ways. In these cases, the technical author doesn't want to know what great ideas the designer has had; they want to keep the designer under control.

▷ Scenarios and their use are discussed in box 13.7, "Scenarios" (p. 480).

If we worry too much about errors in any documents, we won't get around to writing them until the device design is stabilized—probably only a few weeks before it is shipped! It'll mean that the documentation is written, finally, in a hurry and any insights the authors get from writing it (or trying it out on users) will be lost to the design process. Instead, as I've emphasized, authors should start as soon as possible—with no, or as little as possible, wasted effort.

One approach is to use tools to validate documentation, or to generate a few choice phrases accurately. All our previously considered approaches have relied on automatic design tools generating the bulk of the documentation text, with perhaps the technical authors refining it from there. The idea is to provide "handles" onto the device design specification so that bits of the documentation can either be checked or generated directly.

Here are some examples, developing from the framework. Imagine some uninspired but correct text like the following:

> To get from the device Power 1 to Power 2:
> Press [Time]
> Press [Power]

▷ In section 11.5 (p. 393) shows how to generate correct text like this using the framework.

Let's suppose the technical author wants to rephrase this sort of instruction. Here there are many ways of doing it—for simplicity, we'll assume the author is writing in HTML:

- First, the technical author can write something simple like this:

```
If you want to increase the power setting,
<script> getFrom("Power 1", "Power 2"); </script>.
```

In this method, the JavaScript inserts some accurate and reasonably written English, for instance, `press <span class=key>Time</span> then press <key>Power</span>`. (Note that the author has written a period after the closing `</script>`.) If you are keen, the function `getFrom` could have some more parameters to refine its style, so that the technical author has more flexibility in phrasing.

- There is no need to use the framework *just* to say how to use the device, as it can provide all sorts of other information. Suppose there is a more basic function than `getFrom`, called `dataGetFrom` that returns a list or array of button presses. With this function, the technical author could write arbitrary JavaScript to format the information derived from the device specification. The author could write things like this:

```
Don't bother trying to increase the power setting, because it takes
<script>
  document.write(dataGetFrom("Power 1", "Power 2").length);
</script> presses!
```

That sort of rash comment requires checking—because the designer might fix the problem of taking too many key presses to do something, and then it'd be silly to say "Don't bother to do something because it takes 1 presses!"

▷ The bad spelling and grammar can be fixed easily; see the discussion of the `plural` function in section 4.5 (p. 105).

- If the boilerplate approach seems too restrictive, the author could write like this:

```
If you want to increase the power setting,
press <span class=key>Time</span> then
<span class=key>Power</span>.
<script>
  check("Power 1", "Power 2", ["Time", "Power"]);
</script>
```

… now the JavaScript is being used to check that what the author wrote is correct, but it is not generating the text itself.

If you are keen, you could make the function `check` a lot more sophisticated and flexible; for instance, it could take regular expressions instead of, as here, a strict sequence that has to be exactly right, or flagged as wrong.

- All the examples above gave the author warning if the designer had changed the design specification. The same ideas can give the designer warnings too. Here's one suggestion:

```
A great feature of our oven is that it's
easy to increase the power setting.
<script>
  if( dataGetFrom("Power 1", "Power 2").length > 2)
    alert("The easy-power feature must be restored!");
</script> presses!
```

These examples used cascading style sheets to format how the keys or key presses look. Here's a very simple style sheet, to show how it might be done so that all keys get a simple but consistent format:

```
<style type="text/css">
  span.key { font-family: sans-serif; border-style: outset;
             color: white; background-color: silver;
           }
</style>
```

This will give you consistent, light gray buttons with white sans serif text (text like this) and a simple 3D effect that makes each button stand out. You put this style sheet in the HTML head element, that is, anywhere between `<head>` and `</head>`. If you make any changes to it, every key described in the manual will change its format. As usual for cascading style sheets, this is a very useful separation of style from content—the author can decide in one place what every key will look like—and using the framework has also helped you separate the development of the good English from the checking of its accuracy and correspondence with the device specification.

11.7 Using real people to help design

Real people do not work randomly, like gnomes. So we should modify the simulator to record what people really do.

▷ Section 11.1 (p. 369) mentioned the difficulties of using real people to help evaluate a system design—but once gnomes and other techniques have done an overall evaluation, people can be used very effectively for targeted assessments of a design.

We could ask people to try out tasks, such as getting from one power level to another. We keep track of each button press in each state. At the end of an experiment, we can then print out a summary like this:

Nobody tried to press [Clock] when in state Power 2
Nobody tried to press [Quick defrost] when in state Power 2
Nobody tried to press [Time] when in state Timer 2
Nobody tried to press [Time] when in state Power 1
Nobody tried to press [Clear] when in state Power 2
Nobody tried to press [Power] when in state Power 1
Nobody tried to press [Power] when in state Power 2

With a bit more programming we can answer questions like, "What transitions did the users try that the device isn't designed to support?"

Clock was pressed in state Clock but did nothing
Quick defrost was pressed in state Quick defrost but did nothing
Clear was pressed in state Clock but did nothing
Power was pressed in state Clock but did nothing
Power was pressed in state Quick defrost but did nothing

More sophisticated analysis would likely use a log of the users' button presses, whereas the statistics collected in the function `press` only counted state changes—this throws away the information about which button is pressed, and it also loses information relating to tasks that take more than one button press.

11.7.1 Simulating real people

Our gnomes were blind and uninformed about the look and feel of the devices they were using. In contrast, how humans use a device depends crucially on how it looks and how big it is. The simplest way to make our gnomes more human-like is to use Fitts Law to estimate how long a human would take to press buttons:

```
function trialFitts(d, start, finish)
{ d.state = toStateNumber(d, start);
  var f = toStateNumber(d, finish);
  var lastButton = -1; // initially not a valid button number
  var time = 0;
  while( d.state != f )
  { var b = randomButton(device);
    gnomePress(b);
    if( lastButton != -1 )
      time = time+Fitts(lastButton, b);
    lastButton = b;
  }
  document.write("Gnome takes "+time+" seconds to get from "+
         start+" to "+finish+" states.");
}
```

▷ We give suggestions for exploring design changes based on the Fitts and Hick-Hyman laws in section 9.6.4 (p. 304), where we also define the function `Fitts` we use here.

11.8 Conclusions

This chapter presented two key messages: use random techniques to help solve or provide insight into difficult design problems, and use tools to help generate and maintain good information about a design.

- The state of the art in product design is to ignore mathematics, and—sometimes—to over-emphasize human evaluation of designs. The human impact of a design is of course crucial, but it is very difficult to escape misconceptions and cultural assumptions that everyone in the design process shares. Only complete "gnomic" ignorance can provide unbiased insights.
- Generating good user manuals and other documentation, including sales or legal documents, is more easily and reliably done with help from a design framework.

Microwave ovens were used as a running example. They are simple and easy to understand, but they were only a driving example—with a simple system, we can see the design issues crisply. More complex devices have worse design problems that are even harder to manage well (and they are harder to write clearly about): the need for good design methods is greater than the impression this chapter may have given. The consequences of bad design are often much worse than not being able to cook supper.

Now, at the end of part II, we've got both theoretical and practical ideas and techniques for designing better systems, but there's still a long way to go. Many interactive devices are *very* complex. In part III, next, we change the approach to take a higher-level view: how can we approach real, messy, interaction programming and still be successful?

11.8.1 Further reading

- André J., "Can structured formatters prevent train crashes?" *Electronic Publishing*, **2**(3), pp169–173, 1989. Jacques André describes the typesetting of the train driver's manual, quoted in box 11.3, "Generating manuals automatically" (p. 397).
- Carroll, J. M., *The Nurnberg Funnel: Designing Minimalist Instruction for Practical Computer Skill*, MIT Press, 1990. Jack Carroll is a leading researcher in human-computer interaction, and this is his key book on designing user manuals.
- Chung, K. L., *Elementary Probability Theory with Stochastic Processes*, Springer Verlag, 1979. A good book on probability that also introduces Markov models.
- Knuth, D. E., *Seminumerical Algorithms*, Addison Wesley, 1998, third edition. This, volume 2 of Knuth's classic *The Art of Computer Programming*, is *the* book on random numbers and statistical tests.
- Thimbleby, H. W., Cairns, P., and Jones, M., "Usability Analysis with Markov Models," *ACM Transactions on Computer-Human Interaction*, **8**(2), pp99–132, 2001. This paper goes deeper into Markov models and using them in interaction programming, but without the gnomes.

■ Thimbleby, H. W., "User Interface Design with Matrix Algebra," *ACM Transactions on Computer-Human Interaction*, **11**(2), pp181–236, 2004. See this paper if you want to follow up on the many good uses of matrices in user interface design. The paper provides an introduction to finite state machines, using matrices, and has several worked examples.

There are many computer models of human behavior and performance, including ACT-R and EPIC.

■ Anderson, J. R., and Lebiere, C., *The Atomic Components of Thought*, Lawrence Erlbaum Associates, 1998. ACT is a major cognitive theory, which has been very widely used to study how people perform and behave with interactive systems. Although now somewhat dated, this book remains a very clear presentation of the ACT framework, the system (as it then was) and its philosophy. For more recent work, see act-r.psy.cmu.edu

Human error is an unavoidable subject, and is more immediately relevant to interaction programming:

- Reason, J., *Human Error*, Cambridge University Press, 1990. This is the classic reference about human error—full of sound advice, including not using hindsight after an accident to blame the people who made "the" mistake which is now so obvious they shouldn't have made it. (Jim Reason has written many papers and more recent books as well—as usual, look them up on the web, rather than relying on the printed version of *Press On* being the last word.)
- The Institute of Medicine, *To Err is Human: Building a Safer Health System*, National Academy Press, 2000. This book is a wide ranging discussion of human error and how to manage it, rather than how to redesign devices to try to stop it happening in the first place.